

RUBY OPERATORS

http://www.tutorialspoint.com/ruby/ruby_operators.htm

Copyright © tutorialspoint.com

Ruby supports a rich set of operators, as you'd expect from a modern language. Most operators are actually method calls. For example, `a + b` is interpreted as `a.+(b)`, where the `+` method in the object referred to by variable `a` is called with `b` as its argument.

For each operator (`+` `-` `*` `/` `%` `**` `&` `|` `^` `<<` `>>` `&&` `||`), there is a corresponding form of abbreviated assignment operator (`+=` `-=` etc.)

Ruby Arithmetic Operators:

Assume variable `a` holds 10 and variable `b` holds 20 then:

Operator	Description	Example
<code>+</code>	Addition - Adds values on either side of the operator	<code>a + b</code> will give 30
<code>-</code>	Subtraction - Subtracts right hand operand from left hand operand	<code>a - b</code> will give -10
<code>*</code>	Multiplication - Multiplies values on either side of the operator	<code>a * b</code> will give 200
<code>/</code>	Division - Divides left hand operand by right hand operand	<code>b / a</code> will give 2
<code>%</code>	Modulus - Divides left hand operand by right hand operand and returns remainder	<code>b % a</code> will give 0
<code>**</code>	Exponent - Performs exponential (power) calculation on operators	<code>a**b</code> will give 10 to the power 20

Ruby Comparison Operators:

Assume variable `a` holds 10 and variable `b` holds 20 then:

Operator	Description	Example
<code>==</code>	Checks if the value of two operands are equal or not, if yes then condition becomes true.	<code>(a == b)</code> is not true.
<code>!=</code>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	<code>(a != b)</code> is true.
<code>></code>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	<code>(a > b)</code> is not true.

<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.	(a <=> b) returns -1.
===	Used to test equality within a when clause of a <i>case</i> statement.	(1...10) === 5 returns true.
.eql?	True if the receiver and argument have both the same type and equal values.	1 == 1.0 returns true, but 1.eql?(1.0) is false.
equal?	True if the receiver and argument have the same object id.	1 == 1.0 returns true, but 1.eql?(1.0) is false.

Ruby Assignment Operators:

Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assign value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a

**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	$c ** = a$ is equivalent to $c = c ** a$
------------	---------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------

Ruby Parallel Assignment:

Ruby also supports the parallel assignment of variables. This enables multiple variables to be initialized with a single line of Ruby code. For example:

```
a = 10
b = 20
c = 30
```

may be more quickly declared using parallel assignment:

```
a, b, c = 10, 20, 30
```

Parallel assignment is also useful for swapping the values held in two variables:

```
a, b = b, c
```

Ruby Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

Assume if $a = 60$; and $b = 13$; Now in binary format they will be as follows:

$a = 0011\ 1100$

$b = 0000\ 1101$

$a \& b = 0000\ 1100$

$a | b = 0011\ 1101$

$a \wedge b = 0011\ 0001$

$\sim a = 1100\ 0011$

There are following Bitwise operators supported by Ruby language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	$(a \& b)$ will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	$(a b)$ will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	$(a \wedge b)$ will give 49 which is 0011 0001

~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

Ruby Logical Operators:

There are following logical operators supported by Ruby language

Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(a && b) is true.
	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a b) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(a && b) is false.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a && b) is false.

Ruby Ternary operator:

There is one more oprator called Ternary Operator. This first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation. The conditioanl operator has this syntax:

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Ruby Range operators:

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value and a range of values in between.

In Ruby, these sequences are created using the ".." and "..." range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

Operator	Description	Example
..	Creates a range from start point to end point inclusive	1..10 Creates a range from 1 to 10 inclusive
...	Creates a range from start point to end point exclusive	1...10 Creates a range from 1 to 9

Ruby defined? operators:

defined? is a special operator that takes the form of a method call to determine whether or not the passed expression is defined. It returns a description string of the expression, or *nil* if the expression isn't defined.

There are various usage of defined? operator:

Usage 1

```
defined? variable # True if variable is initialized
```

For Example:

```
foo = 42
defined? foo      # => "local-variable"
defined? $_       # => "global-variable"
defined? bar      # => nil (undefined)
```

Usage 2

```
defined? method_call # True if a method is defined
```

For Example:

```
defined? puts      # => "method"
defined? puts(bar)  # => nil (bar is not defined here)
defined? unpack     # => nil (not defined here)
```

Usage 3

```
# True if a method exists that can be called with super user
defined? super
```

For Example:

```
defined? super      # => "super" (if it can be called)
defined? super       # => nil (if it cannot be)
```

Usage 4

```
defined? yield    # True if a code block has been passed
```

For Example:

```
defined? yield    # => "yield" (if there is a block passed)
defined? yield    # => nil (if there is no block)
```

Ruby dot "." and double Colon "::" Operators:

You call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

The :: is a unary operator that allows: constants, instance methods and class methods defined within a class or module, to be accessed from anywhere outside the class or module.

Remember: in Ruby, classes and methods may be considered constants too.

You need just to prefix the :: Const_name with an expression that returns the appropriate class or module object.

If no prefix expression is used, the main Object class is used by default.

Here are two examples:

```
MR_COUNT = 0          # constant defined on main Object class
module Foo
  MR_COUNT = 0
  ::MR_COUNT = 1      # set global count to 1
  MR_COUNT = 2        # set local count to 2
end
puts MR_COUNT          # this is the global constant
puts Foo::MR_COUNT     # this is the local "Foo" constant
```

Second Example:

```
CONST = ' out there'
class Inside_one
  CONST = proc {' in there'}
  def where_is_my_CONST
    ::CONST + ' inside one'
  end
end
class Inside_two
  CONST = ' inside two'
  def where_is_my_CONST
    CONST
  end
end
puts Inside_one.new.where_is_my_CONST
puts Inside_two.new.where_is_my_CONST
puts Object::CONST + Inside_two::CONST
puts Inside_two::CONST + CONST
puts Inside_one::CONST
puts Inside_one::CONST.call + Inside_two::CONST
```

Ruby Operators Precedence

The following table lists all operators from highest precedence to lowest.

Method	Operator	Description
Yes	::	Constant resolution operator
Yes	[] []=	Element reference, element set
Yes	**	Exponentiation (raise to the power)
Yes	! ~ + -	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
Yes	* / %	Multiply, divide, and modulo
Yes	+ -	Addition and subtraction
Yes	>> <<	Right and left bitwise shift
Yes	&	Bitwise 'AND'
Yes	^	Bitwise exclusive 'OR' and regular 'OR'
Yes	<= < > >=	Comparison operators
Yes	<=> == === != =~ !~	Equality and pattern match operators (!= and !~ may not be defined as methods)
	&&	Logical 'AND'
		Logical 'OR'
	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= { /= -= += = &= >>= <<= *= &&= = **=	Assignment
	defined?	Check if specified symbol defined
	not	Logical negation
	or and	Logical composition

NOTE: Operators with a *Yes* in the method column are actually methods, and as such may be overridden.